공학석사학위논문

메모리 액세스 패턴 기반 DRAM 컨트롤러 디자인

Memory Access Pattern-Aware Memory Controller Design

2017년 8월

서울대학교 대학원
컴퓨터공학부
어 정 윤

메모리 액세스 패턴 기반
DRAM 컨트롤러 디자인

Memory Access Pattern-Aware
Memory Controller Design

지도교수 이 창 건

이 논문을 공학석사 학위논문으로 제출함
2017년 4월

서울대학교 대학원
컴퓨터공학부
어 정 윤

어 정 윤의 석사 학위논문을 인준함
2017년 6월

위 원 장 민 상 렬 (인)
부위원장 이 창 건 (인)
위 원 하 순 회 (인)

초 록

Mixed-criticality systems integrate tasks with various levels of criticality onto a same hardware platform. Critical tasks require tight bounding of worst-case latency at any cost, yet for non-critical tasks it is important to provide high performance as much as possible. From this, a tough design concern arises; how to achieve the conflicting demands of *performance isolation* for critical tasks and *efficient sharing* for non-critical tasks in terms of shared DRAM bandwidth and capacity?

Recently, modern mixed-criticality systems are facing rapid change in workloads. One of the biggest challenges among this is the advent of memory-intensive workloads in line with migration to multicore. Memory-intensive workloads significantly exacerbate contention and interference problems in shared memory resources of multicore architectures. This not only endangers tight bounding of worst-case latency of critical tasks, but also, if not properly addressed, can lead to significant performance penalty and unfairness among non-critical tasks.

In this paper, we take workload-driven approach and propose a novel *workload-aware* memory controller design for mixed-criticality system that can successfully achieve both of the conflicting demands in the presence of memory-intensive workloads. Based on the key observation that *memory access pattern of an application captures major memory requirements of the application*, our memory controller manages shared DRAM as a set of memory access pattern-aware partitions - latency sensitive, locality sensi-

tive, and bandwidth sensitive. Our design allocates bandwidth and capacity customized to each partition's needs. By using bank partitioning and request batching with prioritizing, we guarantee short worst-case latency for critical tasks and high performance and fairness to non-critical tasks.

주요어 : Mixed-Criticality Systems, Memory Controller, Memory Access Pattern

학번 : 2015-22903

Contents

List of Figures

List of Tables

제 1 장

Introduction

Mixed-criticality systems are real time systems where tasks of different criticalities are integrated onto a same hardware platform [1]. The goal of mixed-criticality systems is to guarantee strict safety assurance to critical tasks while providing high performance to non-critical tasks at the same time. These conflicting demands impose a difficult challenge to mixed-criticality system design. For strict safety assurance, it should guarantee *performance isolation* of critical tasks from others. But for high performance, allowing *efficient sharing* of underlying hardware resources is necessary.

With the advent of autonomous vehicles, mixed-criticality systems are facing rapid workload changes. Traditional real-time workloads that mixed-critical systems targeted are memory non-intensive programs which mainly run on computing units and seldom generate memory requests. But recent trends in automotive and avionics industry present a new type of mixed-criticality system that runs not only these traditional real-time workloads, but also *memory intensive workloads that are foreign to it*. For example, as commodity self-driving cars are becoming a reality, cars are being transformed into mixed-criticality systems of extreme type. Self-driving cars run extremely memory intensive workloads such as deep neural network-based object detection applications[2, 3, 4] that have soft real-time constraints.

At the same time, they should run traditional memory non-intensive critical tasks that control and actuate the dynamics of the car's physical system.

For the realization of self-driving cars with sensible cost, it is important to devise an efficient method that extremely memory intensive applications and safety-critical tasks are *integrated into a same hardware platform.* This new type of mixed-criticality system should achieve two goals. For memory intensive low criticality tasks that are sensitive to performance, it should allow efficient sharing of memory resources. For memory non-intensive high criticality tasks that require performance isolation, it should provide strict safety assurance.

Research on memory controller for mixed-criticality systems have focused on ensuring performance isolation for strict safety assurance to critical tasks[5, 6, 7, 8, 9]. However, systematic approaches to efficiently share the underlying DRAM resources while preserving the performance isolation have been largely absent. This becomes especially a serious problem with memory intensive workloads. From the performance isolation point of view, memory intensive workloads worsen the already serious problem of contention and interference in shared memory resources of multi-core architecture[10, 11]. Memory requests from co-runner tasks that arbitrarily interferes with safety-critical tasks threatens. From the efficient sharing point of view, customized allocation and sharing of DRAM bandwidth and capacity based on workload's memory requirements becomes crucial. Without careful allocation and scheduling of bandwidth and capacity, it is well-known that workloads containing both memory non-intensive and memory intensive applications easily suffer from performance degradation and seri-

ous unfairness problems[12, 13, 10, 14].

In this paper, we take a novel workload-driven approach in designing memory controller for mixed-criticality systems. Our design is based on the key observation that memory access pattern captures the application's memory requirements. It explicitly controls bandwidth and capacity needs of each tasks by categorizing tasks into three memory access pattern groups.

제 2 장

Background on DRAM Basics

In this section, we provide background knowledge of modern DRAM architecture that is sufficient in understanding the solution in this paper. For more details, we refer the reader to [15, 16].

2.1 DRAM Architecture and Characteristics

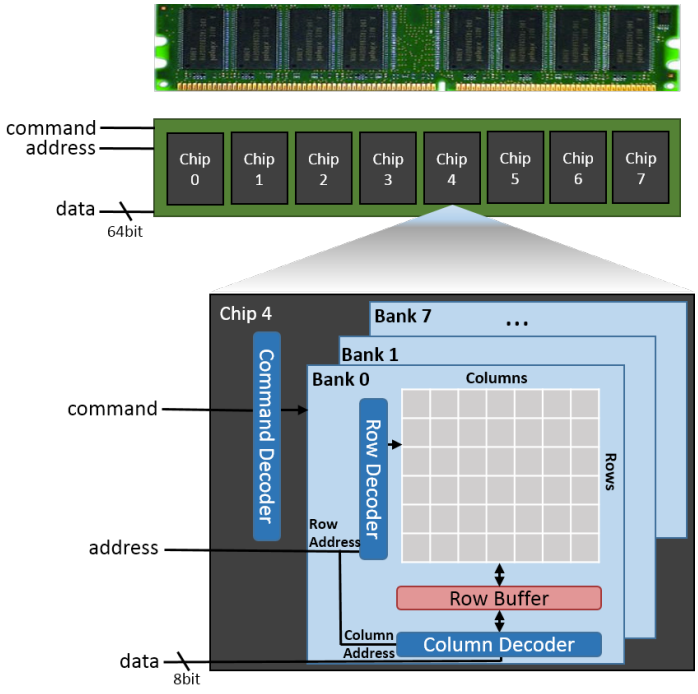

그림 1: DRAM organization

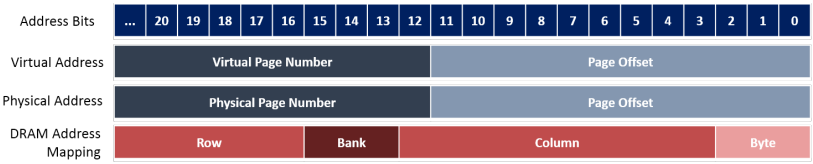

그림 2: Task address to DRAM location mapping

Row Buffer Locality(RBL). Modern DRAM consists of multiple units called *banks*. A DRAM bank is a two-dimensional array. Data are stored in its cells, the interconnection point of a row and a column of the array, as in Figure 1. To read or write data, first the whole row that contains the desired cell should be loaded into a *row buffer* inside the bank. To access data located in another row, the row buffer should be emptied before that row is loaded, which takes additional time. Due to this fact, row buffer acts as an *internal cache of a DRAM bank*. A DRAM bank enjoys cache hit benefit when it is a Row Hit case and suffers from cache miss penalty when it is a Row Miss or a Row Conflict case. In terms of latency, the benefit and penalty can be analyzed as follows(See Table 1 for the DRAM timing parameters and Table 2 for DRAM commands)[11]:

표 1: DRAM timing parameters

Parameters	Symbols	DDR3-1333
DRAM clock cycle time	t_{CK}	1.5 nsec
Precharge latency	t_{RP}	9 cycles
Activate latency	t_{RCD}	9 cycles
CAS read latency	t_{CL}	9 cycles

- *Row Hit*: The request accesses data contained in the row buffer. In

this case, only *RD/WR* command is needed to access the data. Bank access latency is t_{CL}.

- *Row Miss*: Either the row buffer is empty or contains another row. If it is empty, the desired row is first loaded with *ACT* command and accessed with *RD/WR* command. The resulting bank access latency is t_{RCD}+t_{CL}. If it contains another row, it is flushed, loaded, and accessed with *PRE, ACT, RD/WR* commands. Bank access latency is t_{RP}+t_{RCD}+t_{CL}

Bank-Level Parallelism(BLP). Banks operate independent of each other, thus access to different banks can be served in parallel. This allows a level of parallelism at the DRAM. *Bank-level parallelism* denotes for the average number of banks to which there are outstanding requests, when the thread has at least one outstanding request[10]. For memory-intensive workloads, exploiting bank-level parallelism to hide DRAM access latency is critical for high average performance. This becomes ever more important as the gap between CPU clocks and DRAM access latency keeps increasing.

2.2 DRAM Memory Controller

Memory Controller Architecture. Modern memory controller largely consists of two parts - *request buffers* and a *request scheduler*(Figure 2). Request buffers are per-bank queues that store incoming DRAM requests. Request scheduler determines the next request that to gain access to DRAM. When threads running on each core generate memory requests, they are

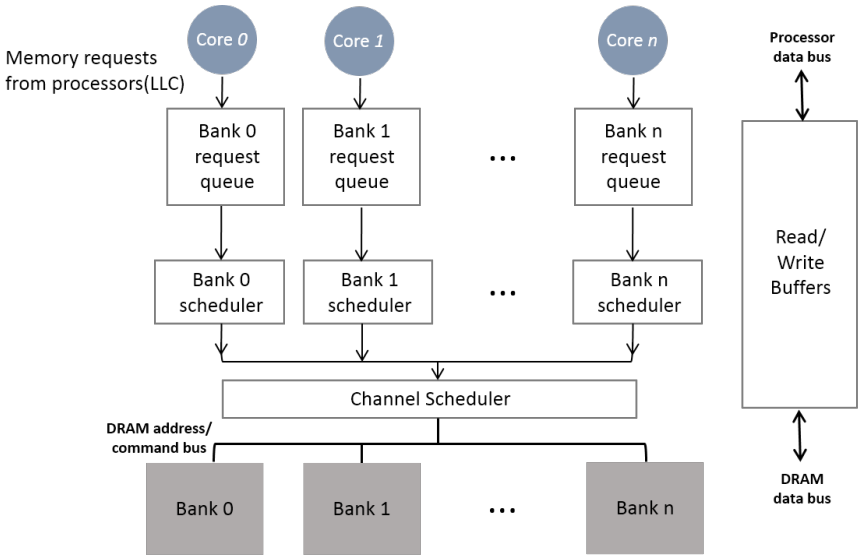

그림 3: Memory controller architecture

sorted by banks in the memory controller arbiter. The scheduler first determines candidates for the next request by checking the status of DRAM banks and buses, possible violation in DRAM timing constraints. These candidates are called *ready* requests. Scheduling policy determines the next request to be served among these candidates.

Memory Scheduling Policy. Modern COTS(Commercial Off-The-Shelf) memory controllers adopt a request scheduling policy called FR-FCFS(First Ready-First Come First Served). Among the *ready* requests, it prioritizes requests that will result in row hit case; if multiple such requests exist, then it applies oldest-first policy among them. FR-FCFS scheduling policy well exploits row buffer locality, and proven to achieve highest average throughput in single-core systems[17, 18]. However, in multi-core systems, multiple threads access the DRAM in parallel as a *shared resource*. In this environ-

ment, it is shown that FR-FCFS policy incurs unfairness problems between threads which harm performance as well.

2.3 Bank Partitioning

Bank partitioning is an OS-level mechanism that physically isolates a set of DRAM banks from the rest of the DRAM, thus eliminating interference between threads which access different dedicated parts of the DRAM. Modern OS uses virtual memory that maps thread's virtual address to physical address. Physical address contains *bank bits*, which designate the DRAM bank that the data with this address is stored. By modifying the virtual-physical address mapping, we can allocate only pages whose physical address with specific bank bits. This results in dedicated *bank partitions*. By wisely creating and allocating dedicated bank partitions of various size to threads in the workload, we can not only achieve performance isolation but also efficient utilization of DRAM bandwidth as a shared resource by providing the right amount of bank-level parallelism.

2.4 Memory Access Patterns

There are various ways of categorizing memory access patterns[19, 10]. Among these, we follow the one introduced in [10], hence it is based on DRAM as a shared resource in multi-core machines and well captures memory behavior of memory-intensive workloads.

In [10], a thread's memory access pattern can be of three types; *latency-sensitive(memory non-intensive), locality-sensitive, bank-level parallelism-*

sensitive threads.

Latency-sensitive threads. Latency-sensitive threads are also memory non-intensive threads. Since latency-sensitive threads spend most of their time running on CPU, memory stall time claims large portion of slowdown compared to memory-intensive threads. These threads issue memory requests sparsely, but the ***latency of each request is critical*** to the thread's performance.

Locality-sensitive threads. Locality-sensitive threads are memory-intensive threads that enjoy high row buffer locality. An intuitive example application of this type is a program that *sequentially* accesses a large array. This type of threads ***require relatively fewer banks compared to BLP-sensitive threads, but guaranteeing performance isolation is much more critical*** than them. If co-runner threads *share* and freely access locality-sensitive threads' banks which will arbitrarily flush and reload the row buffer, high locality that ensures short access latency are destroyed.

BLP-sensitive threads. BLP(bank-level parallelism)-sensitive threads are memory-intensive threads that exhibit high bank-level parallelism. Example application would be a program that *randomly* accesses a large array. Performance of this type is very sensitive to the number of banks that it can use, which are not necessarily dedicated, private banks. Thus, from the performance point of view, ***guaranteeing enough number of banks is more important than providing performance isolation.*** Even if co-runners that *share* the BLP-sensitive threads' banks, loss in row buffer locality is negligible compared to the gain from the increased number of banks - increased amount of parallelism.

제 3 장

Observation

In this section, we explain our observations that lead to key features of workload-aware memory controller design.

Observation 1. Request batching enables *both* preserving bank-level parallelism and tight bounding of worst-case latency.

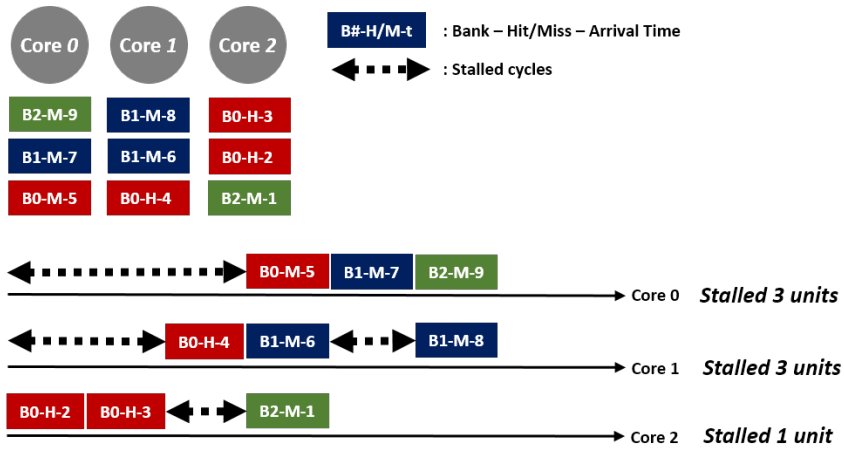

그림 4: FR-FCFS, blindly favoring row buffer locality

Request batching is shown to be effective in guaranteeing parallelism at the bank level[11]. In order to fully exploit the parallelism at the bank level, merely providing multiple banks is not enough. Even if there exist a stream of requests that can be processed by multiple banks in parallel,

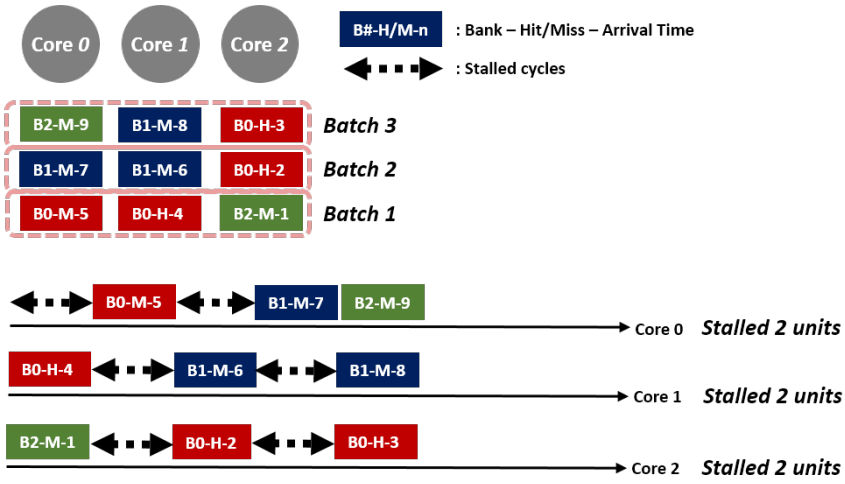

그림 5: Request batching, fair progress of all tasks

requests from other threads can arbitrarily interfere with it in the request scheduler which destroys the potential parallelism(Figure 4).

By forming *a batch of memory requests*, we can construct a flexible granularity that allows us to preserve bank-level parallelism within it[11]. This batch of requests can be thought of as a 'pseudo request' due to its atomicity; requests inside a batch are treated as a single request, so that they're not preempted by any other requests. This enables the preserving of potential parallelism in the request stream which is formed into a batch.

Memory controller that adopts flexible request batching mechanism for non-critical tasks can ensure them to fully exploit available bank-level parallelism of the workload. Our memory controller attempts to maximize the parallelism inherent in the BLP-sensitive tasks by request batching, and maximize the locality inherent in the locality-sensitive tasks at the same time by isolating them from other tasks with bank partitioning.

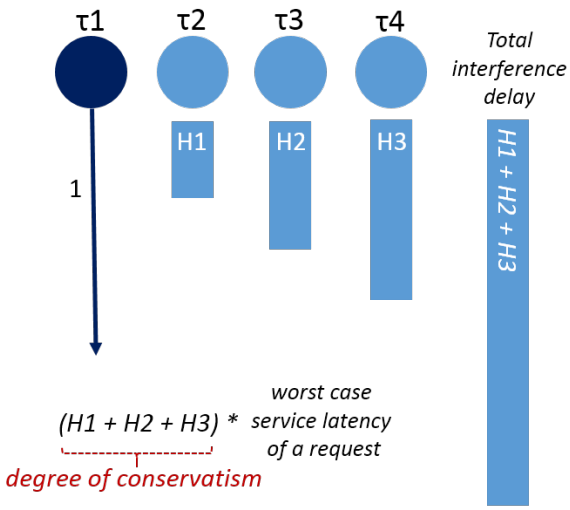

그림 6: Worst-case interference delay under request granularity

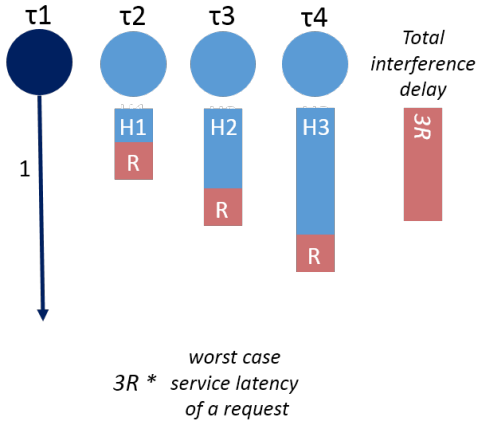

그림 7: Worst-case interference delay under batch granularity

In addition to preserving parallelism, request batching allows us to devise a scheduling policy with greatly reduced pessimism compared with FR-

FCFS scheduling, widely adopted in most commodity memory controllers for performance. Under our memory controller design, worst-case interference delay of tasks in locality-sensitive partition is statically determined *regardless of memory intensity of the co-runner tasks in BLP-sensitive partition*. However, FR-FCFS scheduling aggressively reorders requests such that row buffer hit requests are served earlier than any other requests[17]. Under FR-FCFS scheduling, bounding worst-case interference delay is both complicated due to this reordering and destined to be conservative because maximum possible requests that can be generated by any job of a task are counted as a worst case interference.

Thus, by restricting the maximum number of possible interfering requests as a constant which is multiple of batch size, we can achieve much tighter and simple worst-case interference delay analysis for tasks in locality-sensitive partition.

Observation 2. Memory access pattern-aware bank partitioning maximizes performance and fairness by providing the right amount of DRAM banks.

Recent works in real-time systems field that adopt bank partitioning as a way of performance isolation generally overlooked the fact that the number of allocated banks can significantly affect performance of a task. Banks are equally allocated to all tasks *regardless of their memory access patterns*, focusing only on eliminating inter-core interference and thus achieving performance isolation. However, we observed that the number of banks clearly affects application's performance depending on its memory

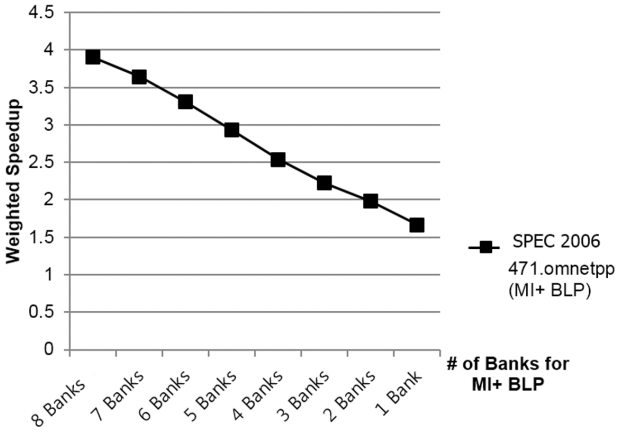

그림 8: MI+ BLP

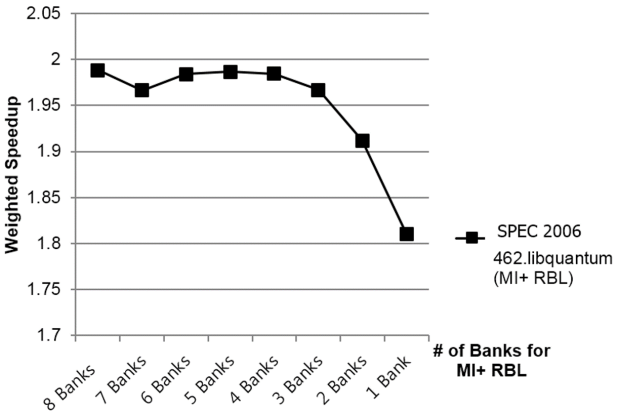

그림 9: MI+ RBL

access pattern(Figure 6). Especially for BLP-sensitive application, whose performance depends on the number of banks that can serve requests in parallel, the number of banks seriously affects performance(Figure 6 (b)). For locality-sensitive application, it was shown that reducing the number of banks doesn't affect performance much, but it definitely needs a small number of private banks. This preliminary result well supports the intuition that,

by allocating capacity(number of banks) and bandwidth(number of memory requests allowed per unit time) according to each application's memory access pattern, much efficient resource usage would be possible, compared with blindly allocating equal amount of banks and bandwidth to each application. Other works[20, 21] also have demonstrated that 8 to 16 banks are enough for an arbitrary application to gain around 90% of its maximum performance.

제 4 장

Memory Access Pattern-Centric Memory Controller Design

4.1 Memory Controller Architecture

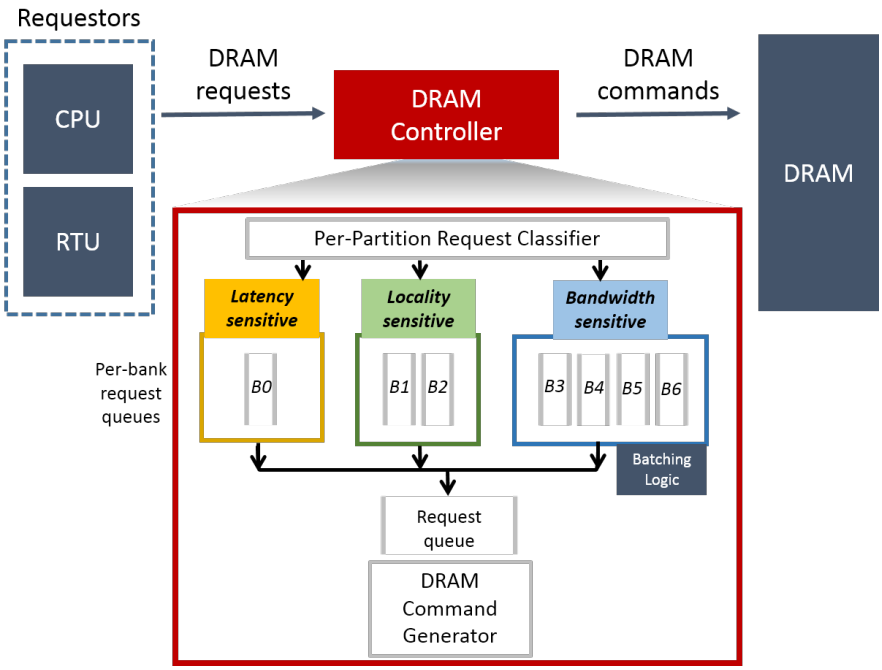

그림 10: Memory access pattern-aware memory controller design

- Algorithm1. Request batching

Our memory controller consists of three *memory access pattern-aware*

partitions - latency-sensitive, locality-sensitive, BLP-sensitive partition. A task in the given workload is included in a partition that best suits its memory access pattern. Each partition provides different capacity and bandwidth resources to its member tasks.

4.1.1 Memory access pattern-aware bank partitioning

For capacity, latency-sensitive and locality-sensitive partitions provide small number of private banks, as tasks of these memory access patterns don't need large amount of banks but require isolation from co-runner tasks on other cores. BLP-sensitive partition provides large number of shared banks because for tasks of this type number of banks that can serve requests in parallel is critical to performance. At the same time, due to low row buffer locality strict isolation is often not needed.

4.1.2 Partition-based prioritization and request batching

For bandwidth, providing right prioritization is important. Requests from latency-sensitive partition gets the highest priority, since tasks of this type seldom generate memory requests but keeping their latency short is critical. Requests from BLP-sensitive partition has the next highest priority, since prioritizing BLP-sensitive requests over locality-sensitive requests is proven to be more fair than vice versa[10]. Requests from locality-sensitive partition has the lowest priority, since they suffer the least fairness degradation from inter-core interference.

For requests from BLP-sensitive partition that use large number of shared banks, we use request batching. As in observation 1, it is important to *effectively guarantee* bank-level parallelism inherent in BLP-sensitive tasks, and merely providing sufficient number of banks is not enough. Request batching unit forms maximum `MarkingCap` number of requests per each core, generating maximum `Number of cores in BLP-sensitive partition * MarkingCap` size batch each time. Next batch is formed only after current batch is completely served, as in [11](Algorithm 1).

4.2 Worst-Case Interference Delay Analysis

We assume the following task model:

$$\tau_i = (C_i, T_i, D_i, H_i)$$

- C_i: WCET of any job of τ_i under single-core environment
- T_i: the minimum inter-arrival time of τ_i
- D_i: relative deadline of τ_i
- H_i: the maximum number of requests generated by any job of τ_i

4.2.0.1 Latency-sensitive partition

For requests in latency-sensitive partition, due to its highest priority it is neither preempted by requests from locality-sensitive partition nor from BLP-sensitive partition. The only potential delay comes from the previous requests sent to the same private bank. Thus, the worst possible delay occurs

when the request just prior to it being serviced from the same bank is a row miss.

- ***Worst case latency for Row Hit***: Read's data transfer takes $CL+BL/2$ and additional 2 cycles for data bus turn-around time[15]. Write's data transfer takes $WL+BL/2$ and additional possible $max(t_{WTR},t_{WR})$ for the data bus turn-around/write recovery time.

  $$L_{hit} = max\{CL+BL/2+2, WL+BL/2+max(t_{WTR},t_{WR})\} \cdot t_{CK}$$

- ***Worst case latency for Row Miss***: Row miss requires all three of PRE, ACT and data read or write commands.

  $$L_{miss} = (t_{RP}+t_{RCD}) \cdot t_{CK} + L_{hit}$$

Iterative response time test for the requests from latency-sensitive partition can be rewritten as below:

$$R_i^{k+1} = C_i + \sum_{\tau_j \in hp(\tau_j)} \lceil \frac{R_i^k}{T_j} \rceil \cdot C_j + L_{miss}$$

4.2.0.2 BLP-sensitive partition

For requests in BLP-sensitive partition, it can be preempted by latency-sensitive partition's requests. And in the worst case, a request can arrive right after the formation of batch ended, thus scheduled to the next batch. Hence, the worst delay it can suffer occurs when it is preempted by the maximum

possible number of memory requests that any job of latency-sensitive partition's task can generate, and then their exist an already-formed batch from BLP-sensitive partition. Since it's the worst case, we assume all of these requests are row miss.

$$R_i^{k+1} = C_i + \sum_{\tau_j \in hp(\tau_j)} \lceil \frac{R_i^k}{T_j} \rceil \cdot C_j + H_{latency-sensitive} \cdot L_{miss}$$

4.2.0.3 Locality-sensitive partition

For requests in locality-sensitive partition, it can be preempted by requests from both latency-sensitive partition and batches of requests from BLP-sensitive partition. In the extreme scenario where BLP-sensitive partition continuously generates requests, requests from locality-sensitive partition suffer from starvation due to the continuous preemption by the batches. To prevent this, we put the limit on the maximum number of batches that can be consecutively served, `MaxConsecutiveBatches`.

The worst delay locality-sensitive partition can suffer occurs when it is preempted by the maximum possible number of memory requests that any job of latency-sensitive partition's task can generate, and then there exist `MaxConsecutiveBatches` number of already-formed batches from BLP-sensitive partition. Here too, we assume all of these are row miss.

$$BatchSize = MarkingCap \cdot BLPPartitionCores$$

$$R_i^{k+1} = C_i + \sum_{\tau_j \in hp(\tau_j)} \lceil \frac{R_i^k}{T_j} \rceil \cdot C_j$$

$$+ (H_{latency-sensitive} + MaxConsecutiveBatches \cdot BatchSize) \cdot L_{miss}$$

제 5 장

Evaluation

5.1 Experiment Setup

We used *Ramulator#*, a cycle-accurate DRAM simulator with representative latency-sensitive(memory non-intensive), locality-sensitive, BLP-sensitive workloads from SPEC 2006[22].

표 2: Characteristics of SPEC 2006 benchmarks used

Benchmark	MPKI	RB Hit Rate	BLP
444.namd	0.33	86.6%	1.27
462.libquantum	50.00	98.4%	1.10
471.omnetpp	22.15	26.7%	3.78

MPKI determines memory intensity, which denotes the number of DRAM requests per kilo instructions. RB Hit Rate denotes row buffer hit rate, which is the number of row buffer hit divided by total DRAM requests. BLP denotes bank-level parallelism and is the number of total banks that received at least one DRAM request when any one of the banks received memory requests.

For *Ramulator#* system setting, we used 3-core where each benchmark run and 4G DDR3-DRAM with one channel, one rank, 8 banks, and 1333MHz memory clock. `MarkingCap` is set to 5, `MaxConsecutiveBatches` is set to 2.

5.2 Performance result of non-critical tasks

We experimented FR-FCFS, FR-FCFS with bank partitioning, and our proposed solution. As the metric for system throughput we used *weighted speedup*[23] and for fairness we used *maximum slowdown*.

$$WeightedSpeedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

$$MaximumSlowdown = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}$$

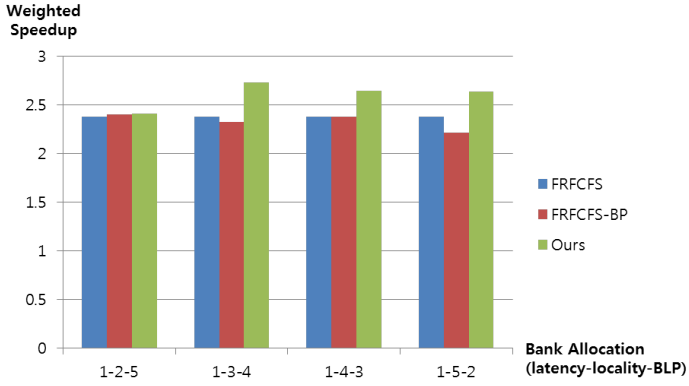

그림 11: Weighted speedup under various bank allocations

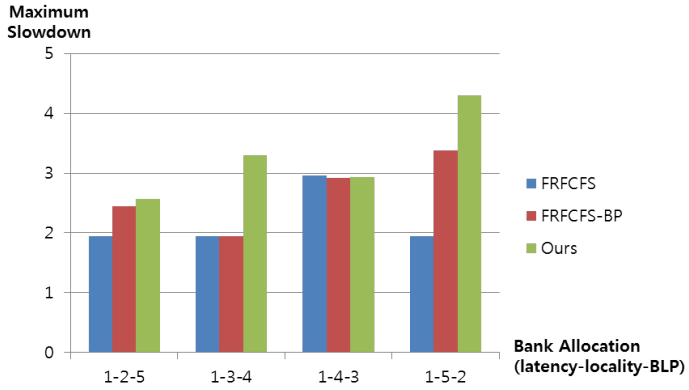

그림 12: Maximum slowdown under various bank allocations

We compared different combination of bank allocation and tested the performance and fairness under the condition.

For performance, we can see that even if banks are allocated against tasks' memory access patterns, our proposed method wins FR-FCFS and FR-FCFS with bank partitioning. The main gain came from the fact that the performance gain of tasks in locality-sensitive partition outgrows the performance loss of tasks in BLP-sensitive partition. This result shows the importance of performance isolation to locality-sensitive tasks - namely, guaranteeing enough number of private banks is very important for locality-sensitive tasks' performance.

On the other hand, when we look at the maximum slowdown in line with this(Figure 11), we can see that fairness is paying the price. Even though the net performance gain increased due to locality-sensitive tasks, BLP-sensitive tasks suffer from severe slowdown as the number of allocated shared banks drops. This is due to the bank allocation ignorant of memory access pattern.

By testing various bank allocation and finding the best trade-off point between performance and fairness, our memory access pattern-aware memory controller can provide both performance isolation and efficient sharing to tasks with various memory access patterns' needs.

제 6 장

Related Work

Bank Partitioning. Bank partitioning has been widely researched as a software-level solution for inter-core interference at main memory level. [20, 24] proposed a way to profile bank address of commodity hardware, and implemented bank partitioning as a linux kernel page allocator. [25, 26] adopted bank partitioning as a way of main performance isolation mechanism for real-time tasks running on COTS(commercial off-the-shelf) hardware.

Bounding Memory Interference Delays. Tightly bounding worst-case interference delays at main memory have gained importance along with migration to multicore architecture. [15] developed worst-case response time analysis that captures interference delay due to reordering effect of row buffer hit requests, which is adopted in most commodity memory controller hardware today[17]. [27] extended [15] and took various factors that define memory-level parallelism such as number of msrhs into consideration.

Memory Controller Design for Mixed-Criticality Systems. [28] first suggested bank privatization method, which achieves predictable bank access latency by scheduling accesses to each banks in a TDM way and generating DRAM commands in a predetermined way. [29] took a step forward and proposed a run-time reconfigurable memory controller that takes suit-

able trade-offs between bandwidth, response time and power. [30] provided a shared-resource abstraction for predictable and composable memory. [31] adopted bank privatization to have bounded worst-case latency for critical tasks, and FR-FCFS aimed at maximized performance for low critical tasks. [32] viewed DRAM as a set of virtual devices and provided a partitioning mechanism to run mixed critical workloads to each virtual device.

While research on memory controllers for mixed-criticality system mainly focused on performance isolation for critical tasks[28, 29, 30, 32], [31] first suggested a way of providing maximized performance to low critical tasks, but overlooked known issues with FR-FCFS scheduling algorithm in terms of fairness; i.e. it blindly prioritizes locality-sensitive workloads, hence bandwidth-sensitive workloads suffer unfair performance degradation.

Memory Access Pattern-Aware Memory Controller Scheduling. [10] first proposed several criteria for categorizing memory access patterns of a program such as memory intensity, row buffer locality and bank-level parallelism. [11] showed that request batching is effective in exploiting bank-level parallelism of programs and proposed a way of achieving trade-off between row buffer locality and bank-level parallelism.

제 7 장

Conclusion

In this paper, we introduced a memory controller architecture that can achieve both performance isolation and efficient sharing of resources, which are two compelling goals of mixed-criticality systems. Our design guarantees strictly bounded worst-case latency for critical tasks and maximizes performance by exploiting locality- and bandwidth-sensitivity for low critical tasks. For future work, we're planning to develop a runtime that analyzes each low critical, memory intensive workload and parses a program into locality-sensitive and bandwidth-sensitive *blocks*. Programs from real world workloads usually consist of locality-sensitive *parts* and bandwidth-sensitive *parts*, while program itself as a whole is often difficult to be clearly classified as locality-sensitive or bandwidth-sensitive. By parsing each programs into *blocks* that well fit the underlying partitions, much more efficient sharing of resources would be possible.

참고 문헌

[1] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[2] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 91–99, Curran Associates, Inc., 2015.

[3] Y. Li, K. He, J. Sun, *et al.*, "R-fcn: Object detection via region-based fully convolutional networks," in *Advances in Neural Information Processing Systems*, pp. 379–387, 2016.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 779–788, 2016.

[5] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, (New York, NY, USA), pp. 99–108, ACM, 2011.

[6] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration.," in *DATE*, pp. 851–856, IEEE, 2011.

[7] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time sdram controller for mixed time-criticality systems," in *Hardware/Software Codesign and System Synthesis*

(CODES+ISSS), 2013 International Conference on, pp. 1–10, IEEE, 2013.

[8] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pp. 317–326, IEEE, 2015.

[9] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling dram memory accesses for multi-core mixed-time critical systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pp. 307–316, IEEE, 2015.

[10] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 65–76, Dec 2010.

[11] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2008.

[12] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 146–160, Dec 2007.

[13] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, Jan 2010.

[14] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact

of inter-application interference at shared caches and main memory," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 62–75, ACM, 2015.

[15] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 145–154, April 2014.

[16] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, (New York, NY, USA), pp. 128–138, ACM, 2000.

[18] S. Rixner, "Memory controller optimizations for web servers," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 355–366, IEEE Computer Society, 2004.

[19] Y. Zhou and D. Wentzlaff, "Mitts: Memory inter-arrival time traffic shaping," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 532–544, IEEE Press, 2016.

[20] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 367–376, ACM, 2012.

[21] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning," in *2014 IEEE 20th International*

Symposium on High Performance Computer Architecture (HPCA), pp. 344–355, Feb 2014.

[22] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[23] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, (New York, NY, USA), pp. 234–244, ACM, 2000.

[24] L. Liu, Z. Cui, Y. Li, Y. Bao, M. Chen, and C. Wu, "Bpm/bpm+: Software-based dynamic memory partitioning mechanisms for mitigating dram bank-/channel-level interferences in multicore systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, p. 5, 2014.

[25] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 155–166, April 2014.

[26] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "Wcet(m) estimation in multi-core systems using single core equivalence," in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 174–183, July 2015.

[27] H. Yun, R. Pellizzon, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 184–195, July 2015.

[28] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 99–108, Oct 2011.

[29] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time sdram controller for mixed time-criticality systems," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–10, Sept 2013.

[30] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *2011 Design, Automation Test in Europe*, pp. 1–6, March 2011.

[31] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 317–326, April 2015.

[32] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 1–10, Aug 2014.

요약 (국문 초록)

혼합 크리티컬리티 시스템(Mixed-criticality system)이란 서로 다른 레벨의 크리티컬리티를 갖는 태스크들을 단일한 하드웨어 플랫폼에 수행하는 시스템이다. 크리티컬 태스크의 대기시간(latency)의 정확한 최악값 보장을 최고의 목표로 하나, 그 외의 태스크들의 경우 최대한도의 성능 향상을 이끌어내는 것이 중요하다. 즉 DRAM 대역폭과 용량 면에서 성능 분리(performance isolation)과 효율적인 자원 공유라는 두 가지 상충되는 목표를 동시에 달성해야 한다. 최근 혼합 크리티컬리티 시스템은 빠른 워크로드 변화를 맞고 있다. 이러한 경향에서 주목할 만한 문제는 멀티코어 아키텍처의 부상과 함께 대두된 메모리 집약적인 워크로드이다. 멀티코어 아키텍처 하에서 메모리 집약적인 워크로드는 공유자원으로 쓰이는 DRAM에서의 메모리 간섭 문제를 악화시킨다. 이는 크리티컬 태스크의 대기시간의 정확한 최악값 보장을 위협할 뿐 아니라 그 외의 태스크들에게 큰 폭의 성능 및 공정성 문제를 야기한다. 본 논문에서는 워크로드 중심 접근법을 시도하여, 메모리 집약적인 워크로드가 있을 때 위의 두 상충하는 목표를 달성하기 위한 워크로드 중심적인 혼합 크리티컬리티 시스템을 위한 메모리 컨트롤러 디자인을 시도한다.